JAVA NOTES

DATA STRUCTURES

Terry Marris August 2001

16 BINARY SEARCH TREES

16.1 LEARNING OUTCOMES

By the end of this lesson the student should be able to

- draw a diagram showing a binary search tree resulting from inserting a given sequence of objects
- explain how to search through a binary search tree
- implement and test a dynamic binary search tree data structure

16.2 PRE-REQUISITES

The student should be comfortable with binary search §5, linked lists §14 and recursion §15.

16.3 INTRODUCTION

In §13 and §14 we looked at linked lists. The problem with a linked list is that we need to search to the end of the list before we can establish that some item is not in the list. We can speed up the search if we apply a binary search method to a collection of nodes organised in a hierarchical and ordered way.

16.4 ORGANISING THE NODES

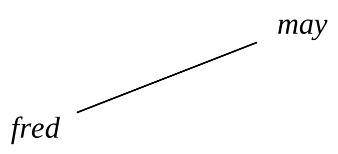
Perhaps the easiest way to demonstrate the ordering is by example. We shall construct a binary search tree using following sequence of objects:

may, fred, sue, daxa, eva, tom, nita

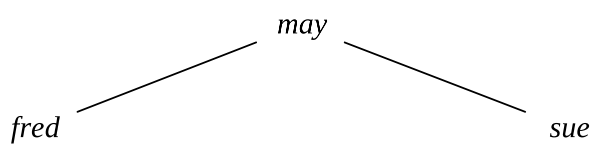
We start with *may*. *may* is positioned at the top.

may

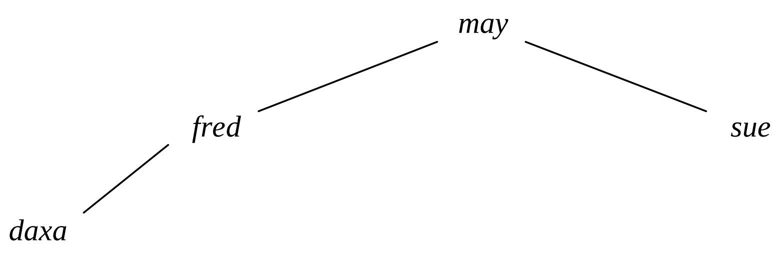
fred comes *before may* in dictionary order. So we place *fred* to the *left* of *may*.



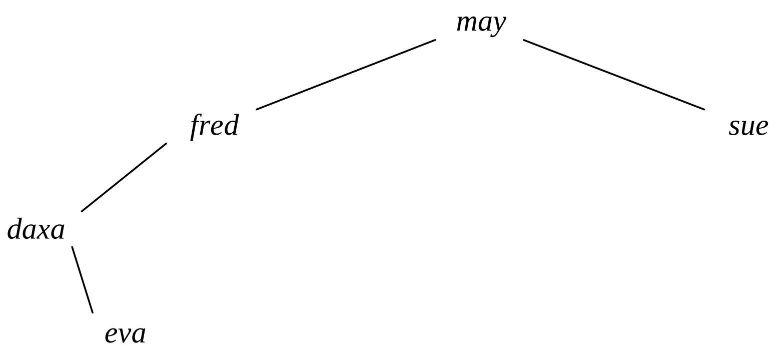
sue comes *after may* in dictionary order. So we place *sue* to the *right* of *may*.



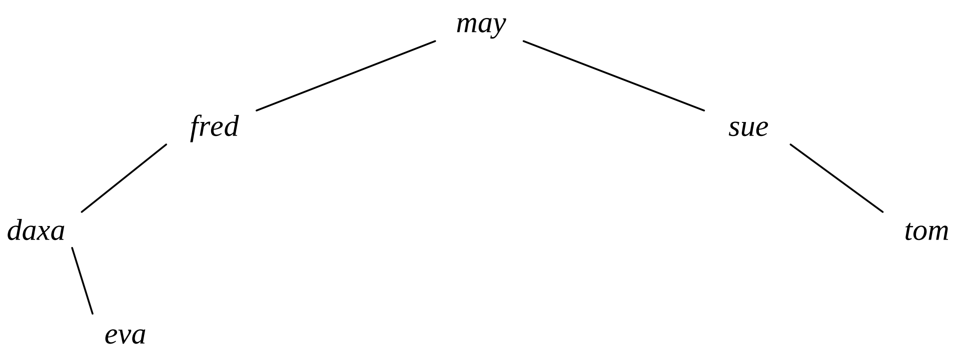
daxa comes before *may* in dictionary order, so we go left and reach *fred*. *daxa* comes before *fred* and so we place *daxa* to the left of *fred*.



eva comes before *may* and before *fred*, but comes after *daxa* in dictionary order. So we place *eva* to the right of *daxa*.



tom comes after *may* and after *sue* in dictionary order. So we place *tom* to the right of *sue*.



nita comes after *may* but before *sue* in dictionary order. So we place *nita* to the left of *sue*.

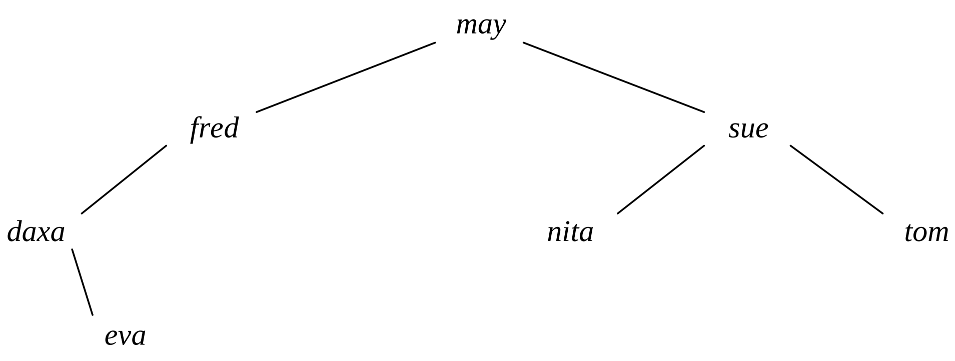


Figure 16.1 *A Binary Search Tree*

16.5 NOTATION

A binary tree has just one root node and several terminal nodes.

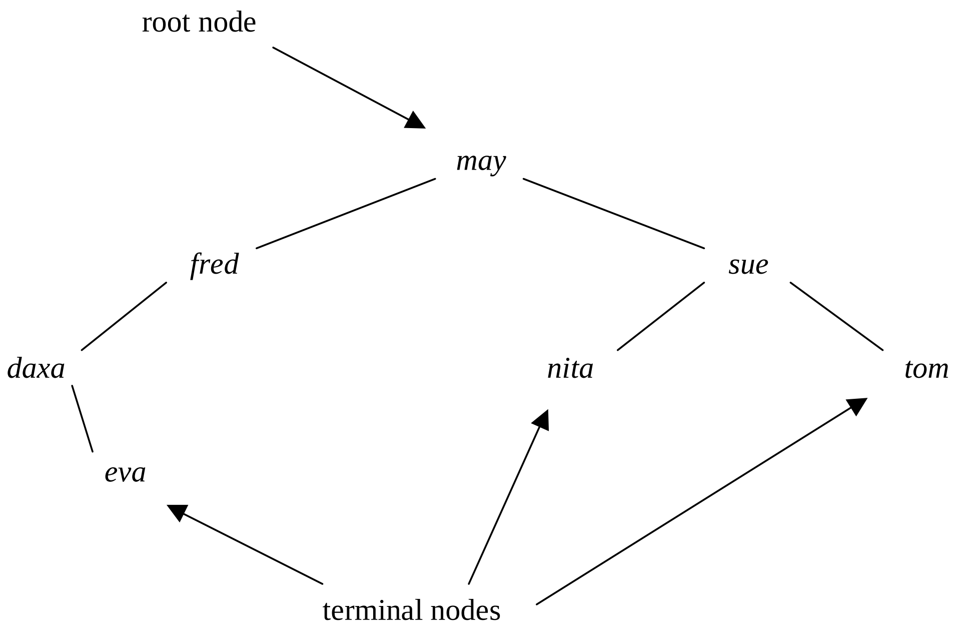


Figure 16.2 *A Binary Tree has Nodes*

A node has zero, one or two children.

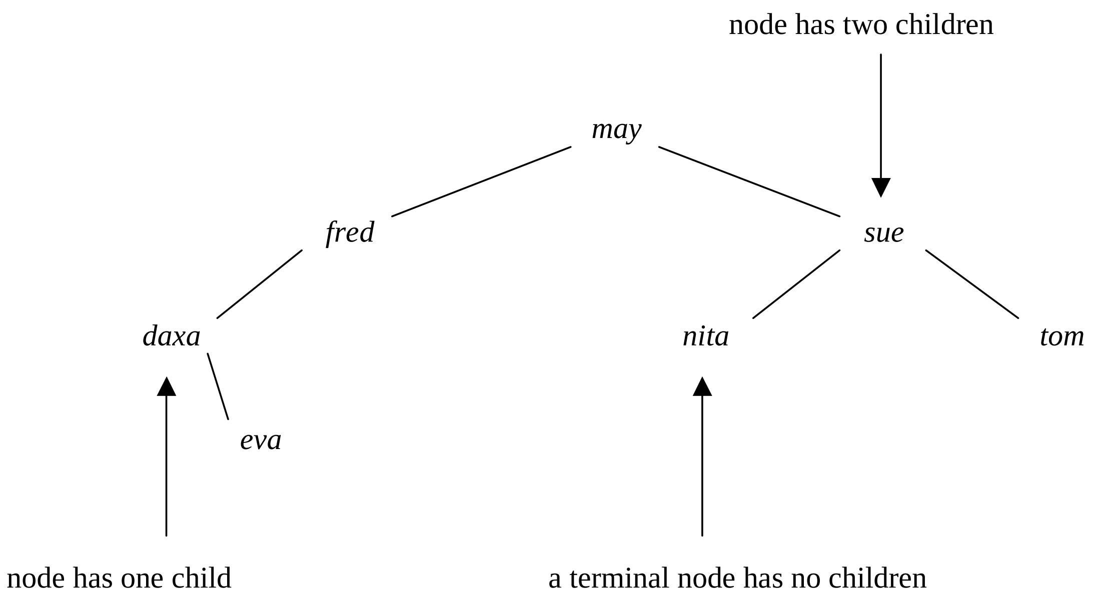


Figure 16.3 *A node has up to two children*

Each node (except the root node) has a parent node. *sue* is the parent node to both *nita* and *tom*.

A binary tree is either empty or it has a root node together with two binary trees called the left subtree and the right subtree. Notice the recursive definition..

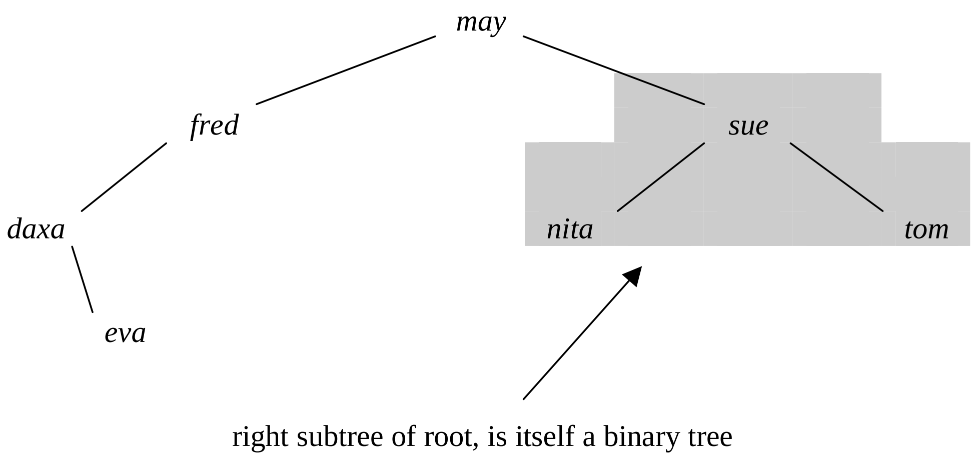


Figure 16.4 *A binary tree may have subtrees*

A binary search tree is a binary tree in which each node has a key data field. All keys in the left subtree come before (e.g. in alphabetical order) the key in the root. All keys in the right subtree come after the key in the root. These keys are unique - you cannot have repetitions.

16.6 NODES

A node has three fields, a data field and two link fields named *left* and *right*. The *left* and *right* fields refer to the node's left and right subtrees.

```
class Node {
  Object data;
  Node left = null;
  Node right = null;

  Node(Object data)
  {
    this.data = data;
  }
}
```

Initially, *left* and *right* both refer to nothing; *data* contains a reference to the data object.

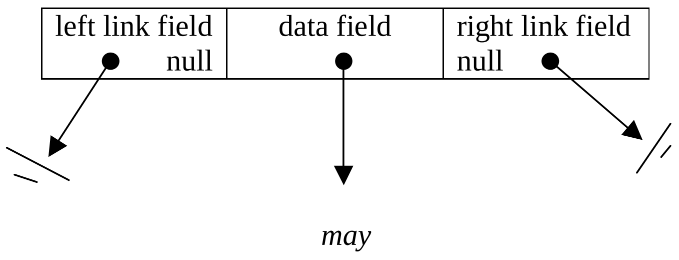


Figure 16.5 *A Node has three fields*

16.7 BINARY SEARCH TREE

A binary search tree has two fields. *root* refers to the root node. *size* contains the number of nodes in the tree. Initially, a tree has no nodes and so *root* contains *null* and *size* is zero.

```
public class BinarySearchTree {
     ...
  private Node root = null;
  private int size = 0;


  public BinarySearchTree()
  {
  }
```

16.8 INSERT A NODE

We start with an empty tree.

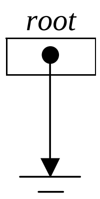


Figure 16.6 *An Empty Tree*

To insert a node into an empty tree is easy: we create a new node containing *sue*, assign it to *root*, increment *size* and return *success*.

```
public String insert(Object obj)
{
  if (root == null) {
    root = new Node(obj);
    size++;
    return "success";
  }
```

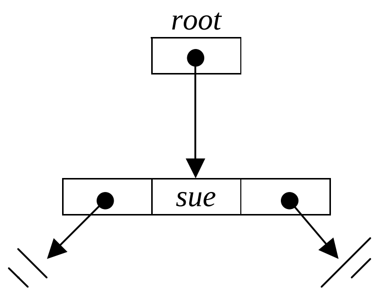


Figure 16.7 *A Binary Tree with One Node*

If the tree is not empty, we make a call to *insert(Node, Object).*

```
  public String insert(Object obj)
  {
    if (root == null) {
      root = new Node(obj);
      size++;
      return "success";
    }
    return insert(root, obj);
  }
```

```
private String insert(Node parent, Object obj)
{
```

At this point *parent* refers to *sue*.

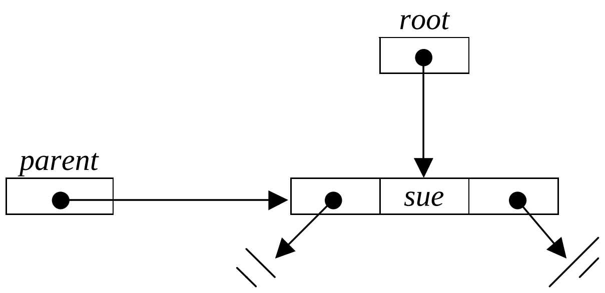


Figure 16.8 *parent.left is null*

We see how to insert *fred.* We start at the *root* node. We search through the tree, going to the left subtree or to the right subtree until we find the right place to insert *fred.* We have two simple cases:

(a) the data item (fred) is already in the tree (duplicates are not allowed, remember)
(b) we reach a parent node that has its left or right link field (whichever is appropriate) containing *null*

```
private String insert(Node parent, Object obj)
 {
   int cmp = ((Comparable)obj).
               compareTo((Comparable)parent.data);

   if (cmp == 0)
     return "failure - duplicate entry";

   if (cmp < 0) {
     if (parent.left == null) {
       parent.left = new Node(obj);
       size++;
       return "success";
     }
    ...
```

At this point, *parent* refers to the *root* node. *parent.left* is *null* and so we have found our insertion point for *fred*. We create the new node containing *fred*, assign it to *parent.left*, increment *size* and return *success*.

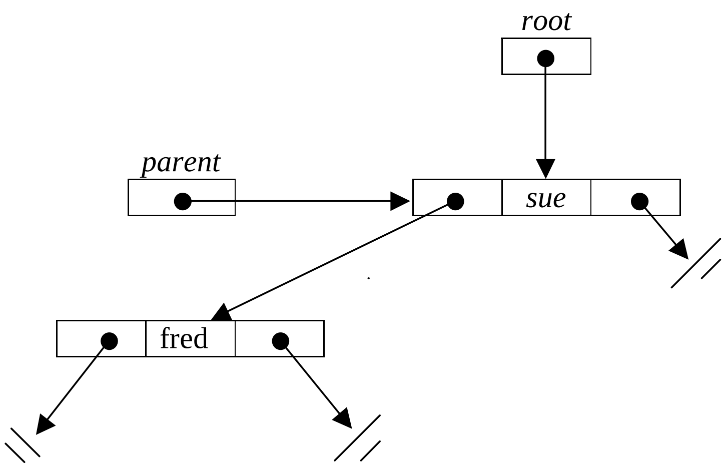


Figure 16.9 *After Inserting fred*

We see how to insert *george*. We start at the *root* node.

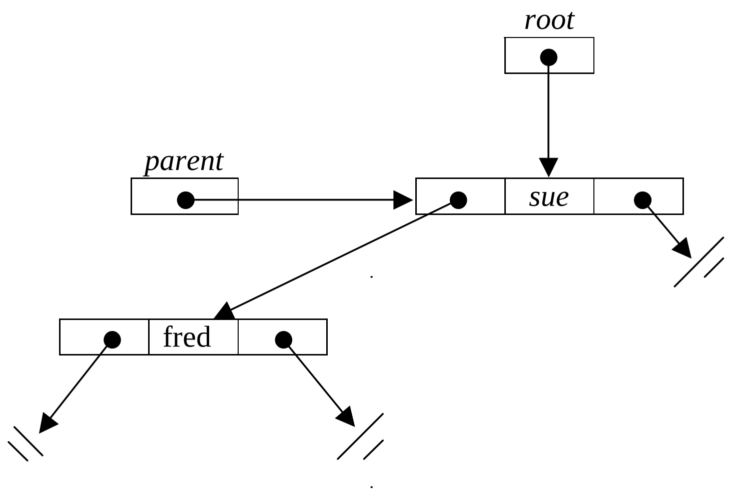


Figure 16.10 *Before Inserting george*

At this point, *parent* refers to the *root* node. *parent.left* is not *null*. So we make a recursive call to *insert(Node, Object)*.

```
private String insert(Node parent, Object obj)
{
  int cmp = ((Comparable)obj).
              compareTo((Comparable)parent.data);

  if (cmp == 0)
    return "failure - duplicate entry";

  if (cmp < 0) {
    if (parent.left == null) {
      parent.left = new Node(obj);
      size++;
      return "success";
    }
    else
      return insert(parent.left, obj);     // recursive call
  }
```

At this point *parent* refers to *fred*.

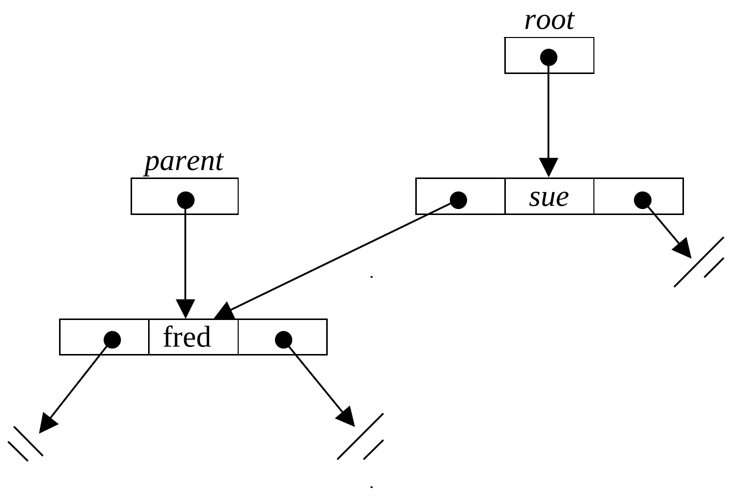


Figure 16.11 *parent.right is null*

Now *parent.right* is null; we have found the insertion point for *george*.

```
private String insert(Node parent, Object obj)
{
  int cmp = ((Comparable)obj).
              compareTo((Comparable)parent.data);

  if (cmp == 0)
    return "failure - duplicate entry";

  if (cmp < 0) {
    if (parent.left == null) {
      parent.left = new Node(obj);
      size++;
      return "success";
    }
    else
      return insert(parent.left, obj);     // recursive call
  }

  // cmp > 0
  if (parent.right == null) {
    parent.right = new Node(obj);
    size++;
    return "success";
  }
  ...
}
```

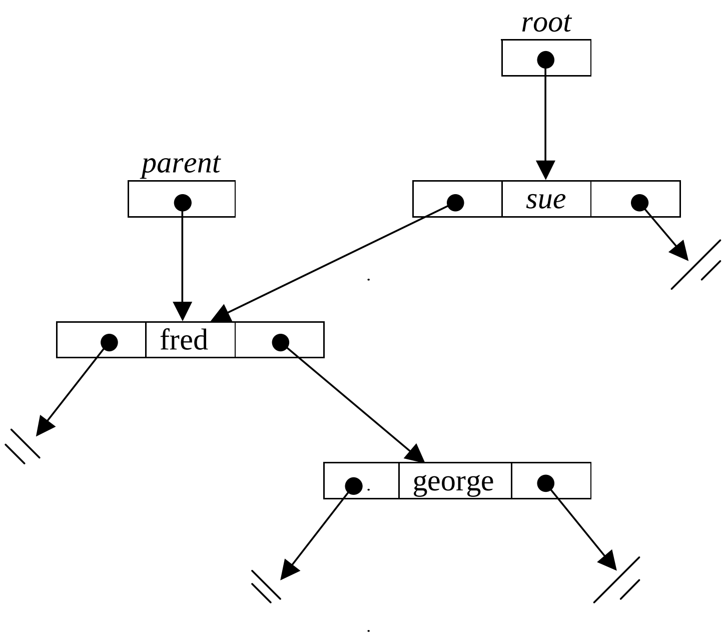


Figure 16.12 *After updating parent.right*

Each recursive call gets you one step closer to a simple case that terminates the sequence of recursive calls.

16.9 IMPLEMENTATION

A *BinarySearchTree* implementation and a small test program are shown below.

```
/* BinarySearchTree.java
   Terry Marris  2 August 2001
*/

public class BinarySearchTree {
  private static class Node {
    Object data;
    Node left = null;
    Node right = null;

    Node(Object data)
    {
      this.data = data;
    }
  }


  private Node root = null;
  private int size = 0;


  public BinarySearchTree()
  {
  }


  public int size()
  {
    return size;
  }
```

```
private String insert(Node parent, Object obj)
{
  int cmp = ((Comparable)obj).
              compareTo((Comparable)parent.data);

  if (cmp == 0)
    return "failure - duplicate entry";

  if (cmp < 0) {
    if (parent.left == null) {
      parent.left = new Node(obj);
      size++;
      return "success";
    }
    else
      return insert(parent.left, obj);
  }

  // cmp > 0
  if (parent.right == null) {
    parent.right = new Node(obj);
    size++;
    return "success";
  }
  return insert(parent.right, obj);
}


public String insert(Object obj)
{
  if (root == null) {
    root = new Node(obj);
    size++;
    return "success";
  }
  return insert(root, obj);
}
```

```
private boolean contains(Node parent, Object obj)
{
  if (parent == null)
    return false;
  int cmp = ((Comparable)obj).
              compareTo((Comparable)parent.data);
  if (cmp == 0)
    return true;
  if (cmp < 0)
    return contains(parent.left, obj);
  // cmp > 0
  return contains(parent.right, obj);
}


public boolean contains(Object obj)
{
  return contains(root, obj);
}
```

```
  public static void main(String[] s)
  {
    BinarySearchTree bst = new BinarySearchTree();

    bst.insert(new String("may"));
    bst.insert(new String("fred"));
    bst.insert(new String("sue"));
    bst.insert(new String("daxa"));
    bst.insert(new String("eva"));
    bst.insert(new String("tom"));
    bst.insert(new String("nita"));
    bst.insert(new String("tom"));  // duplicate entry

    System.out.println("The size of the tree is ... " +
                        bst.size());

    System.out.println("tree contains ann ... " +
                        bst.contains(new String("ann")));

    System.out.println("tree contains may ... " +
                        bst.contains(new String("may")));

    System.out.println("tree contains tom ... " +
                        bst.contains(new String("tom")));

    System.out.println("tree contains ursula ... " +
                        bst.contains(new String("ursula")));
  }
}
```

Output

```
The size of the tree is ... 7
tree contains ann ... false
tree contains may ... true
tree contains tom ... true
tree contains ursula ... false
```

16.10 REVIEW

16.11 FURTHER READING

SCHIFLET A,B Data Structures in C++ pp 559
KRUSE R,L Data Structures and Program Design pp 322
KOFFMAN E,B Problem Solving and Structured Programming in Modula-2 pp 649

Further algorithms, e.g. to delete a node and to maintain a balanced tree are not considered here but may be found in more advanced texts.

This is the last lesson on dynamic data structures. In the next we being our study of persistent data structures, namely files.

16.12 EXERCISES

1 A binary search tree is initially empty. Draw a diagram to represent the tree after the following items have been inserted:

int, char, return, break, float, while, tree, table, binary, network

2 A binary search tree is initially empty. Draw a diagram to represent the tree after the following items have been inserted:

29, 19, 25, 33, 20, 35, 30

3 Where is the node with the least key value in a binary search tree? Where is the node with the largest key value?

4 Thoroughly test the BinarySearchTree implementation described in §9 above.

5 Explain how the *contains(Object)* and *contains(Node, Object)* methods work.

6 Design, write and test methods that, by indexing the nodes in key field order, retrieves a node's data when given an index. Hint: you saw a similar idea in the linked list implementation.